1984

A Guide to the Poker Mapping Preprocessor

Fran Berman

Mike Goodrich

Chuck Koelbel

Bill Robison

Karen Showell

Report Number:
84-488

TR# 488

A Guide to the Poker Mapping Preprocessor

Fran Berman
Mike Goodrich
Chuck Koelbel
Bill Robison
Karen Showell

The Poker Mapping Preprocessor or *Prep-P* is a program for mapping large-sized parallel algorithms onto a 64 PE CHiP architecture [S1]. Prep-P takes as input a graph in a special adjacency list format for one of a given class of interconnection networks. Associated with the input graph are programs written in XX [S2] for each of the nodes in the graph. Prep-P takes the input graph and program codes, contracts and lays out the graph on a 64 PE CHiP architecture with corridor width 1, and multiplexes the code (if needed). The multiplexing is done by concatenating the program codes for each of the nodes mapped onto a single PE and executing their instructions in a round-robin fashion. The result of Prep-P is assembled code ready to be run on a 64 PE lattice by the Poker system [S3]. In this document we describe the components of the program and their use.

Prep-P may be initiated in its entirety by calling

/usr/fdb/preprocessor/bin/Prep-P <filename>

which executes a shellscript calling the individual components of Prep-P in the following order:

Phase1
Phase2
XX

if the graph has > 64 nodes:	*if the graph has ≤ 64 nodes:*
contract	layout
layout	genPN
premux	genCN
StoA51	
As51	
postmux	
HxToO	

This can be done by adding */usr/fdb/preprocessor/bin* to your path, creating a file *<filename>* in the input format given below and typing

Prep-P <filename>

Unless otherwise specified, all programs in this document can be found in */usr/fdb/preprocessor/bin*.

Input

Input to Prep-P consists of a graph type identifier, a specification of the minimum node number and the number of nodes in the graph, and a list of procedures associating a program code identifier and vertex adjacencies with each of the nodes in the graph. XX program codes associated with the

program code identifiers are given as separate files, each under the name of its own program code identifier.

As an example, say we wish to input the binary tree illustrated in Figure 1.

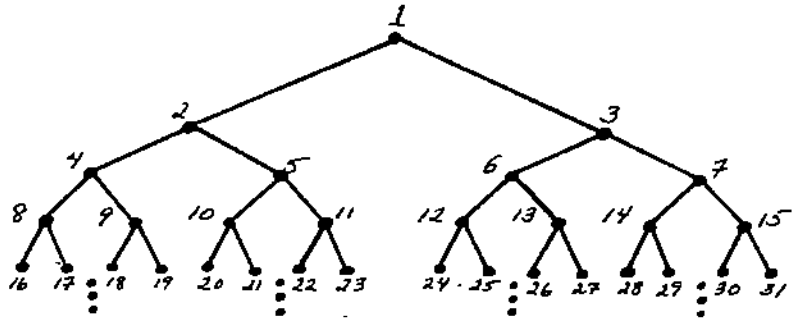


Figure 1
A binary tree with 127 nodes.

We create a file called *graph* (the name of the file can be anything). If the tree has XX program code with identifier ROOT associated with the root, XX program code with identifier ANCES associated with each non-root interior node, and XX program code with identifier LEAF associated with each of the leaves, the specification of the graph in *graph* might appear as follows:

```
tree
nodemin = 1
nodecount = 127
procedure ROOT
  nodetype : {i == 1}
  port RSON : {2*i}
  port LSON : {2*i+1}
procedure ANCES
  nodetype : {i> 1 && i< 64}
  port PARENT : {i/2}
  port RSON : {2*i}
  port LSON : {2*i+1}
procedure LEAF
  nodetype : {i> 127/2}
  port PARENT : {i/2}
```

The first 3 lines of the file specify the graph type, minimum node number and number of nodes. The procedure list is a specification of the adjacencies together with an association of program code identifiers with each node of the graph. In each of the procedures, all the nodes associated with the XX program code of the given program code identifier are given in a boolean expression by *nodetype*. In addition, the adjacency structure of these nodes is given by listing their port names along with an arithmetic expression of the numbers of their incident nodes. The port names in the adjacency structure must correspond to the port names referenced in the XX program code for the given program code identifier.

For the general case, the grammar rules for input specification are given in Figure 2. The function, precedence and associativity of operators are exactly as in the C programming language, however all operations yield integer results. The arithmetic and boolean expressions may only include the variable *i* and integer constants. Comments may appear anywhere in the input file and begin with /* and end with */.

```
<graphspec> :   <decl> <proclist>
<decl> :        <gtype> <nodespec>
<gtype> :       /* equation is the nodecount for some integer k */

                'tree'      /* binary tree              (2**k)-1 */
                'shuffexch' /* shuffle exchange           2**k */
                'ccc'       /* cube-connected cycle     k(2**k) */
                'bincube'   /* binary n-cube              2**k */
                'mesh'      /* mesh                       k**2 */
                'hexmesh'   /* hexmesh                    k**2 */
                'torus'     /* torus                      k**2 */
                'line'      /* line                          k */
                'loop'      /* loop                          k */
                'felem'     /* finite element             k**2 */
                's4pin'     /* four-pin shuffle           2**k */

<nodespec> :    'nodemin' = <INT> 'nodecount' = <INT>
<proclist> :    <proc> | <proc> <proclist>
<proc> :        'procedure' <ID> <nodecl> <portlist>
<nodecl> :      'nodetype' : {<bool_expr>}
<portlist> :    <portdecl> | <portdecl> <portlist>
<portdecl> :    'port' <ID> : {<simp_expr>}

<bool_expr> :   <simp_expr> < <simp_expr>
                <simp_expr> > <simp_expr>
                <simp_expr> <= <simp_expr>
                <simp_expr> >= <simp_expr>
                <simp_expr> == <simp_expr>
                <simp_expr> != <simp_expr>
                ! <bool_expr>
                <comp_bool>

<comp_bool> :   <bool_expr> && <bool_expr>
                <bool_expr> || <bool_expr>

<simp_expr> :   <prim>
                ~ <simp_expr>
                - <simp_expr>
                <simp_expr> | <simp_expr>
                <simp_expr> & <simp_expr>
                <simp_expr> ^ <simp_expr>
                <simp_expr> + <simp_expr>
                <simp_expr> - <simp_expr>
                <simp_expr> * <simp_expr>
                <simp_expr> / <simp_expr>
                <simp_expr> % <simp_expr>
                <simp_expr> << <simp_expr>
                <bool_expr> ? <simp_expr> : <simp_expr>

<prim> :        <ID> | <INT> | {<simp_expr>}
```

Figure 2
Graph input grammar.

As in the C programming language [KR], the *#define* construction may be used at the beginning of the file to allow substitution of symbolic names or constants throughout the file. For example, we could have used

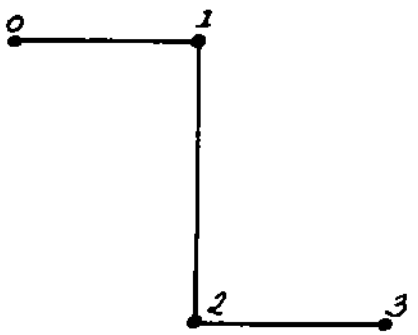
#define nc 127

as the first line in our file and substituted *nc* for every occurrence of 127 in the file (specifically for the *nodetype* specifications in ANCES and LEAF). During the input phase, the graph description file will be filtered through the C preprocessor which strips out comments and expands the symbolic names.

Every vertex in the graph must appear in the *nodetype* of some procedure in the procedure list. In addition, since Prep-P only accepts undirected graphs, if w appears as a port number in a procedure in which v is a *nodetype*, then v must appear as a port number in a procedure in which w is a *nodetype*. If there is no obvious straightforward arithmetic or logical expression by which to abbreviate the adjacency list of a graph, the adjacency list may be input explicitly. For example, if the XX program code identifiers for nodes 0 and 3 are A, and the XX program code identifiers for nodes 1 and 2 are B and C respectively, then the graph specification file for the shuffle-exchange graph in Figure 3b) is illustrated in Figure 3a).

```
shuffexch
nodemin = 0
nodecount = 4
procedure INEND
   nodetype : { i == 0 }
   port EXCH : {1}
procedure INNER
   nodetype : { i == 1 }
   port EXCH : { 0 }
   port SHUF : { 2 }
procedure INNER
   nodetype : { i == 2 }
   port EXCH : { 3 }
   port SHUF : { 1 }
procedure OUTEND
   nodetype : { i == 3 }
   port EXCH : { 2 }
```

a)



b)

Figure 3
Specification file and shuffle-exchange graph.

The input specification for Prep-P requires that all graphs be "full". For example, the shuffle-exchange, 4 pin shuffle and the binary n-cube must have exactly 2^k nodes (for some integer k) and the square mesh, hex mesh, torus and finite element graphs must have k^2 nodes. At this point in time, only graphs with interconnection structures of the following types can be handled by the preprocessor:

complete binary trees
shuffle-exchange
cube-connected cycles (CCC)
binary n-cube
square mesh
hexagonal mesh
torus
finite element graph
line
loop
4 pin shuffle

Each of these graphs must be input with a standard numbering. Graph specification formats and standard numberings for these graphs can be found in the Appendix.

Each XX code associated with the program code identifier must be in a separate file in the same directory from which Prep-P and poker will be called. The name of the file is the program code identifier followed by *.x*. For example, the program codes for ROOT, ANCES and LEAF would have to be located in files with the names *ROOT.x*, *ANCES.x* and *LEAF.x* respectively. The program code must be in XX [S2]. Trace variables may be included but poker can only print out 4 trace variables per PE. This means that only the first 4 trace variables will be printed out for each set of nodes mapped onto an individual PE. We defer discussion of trace variables until later in this document.

The program components of Prep-P which process the input files are called *Phase1* and *Phase2*. *Phase1* checks the syntax of the input files and pipes errors to standard output. *Phase2* checks the number of nodes in the graph input and chooses one of two options. If the number of nodes in the graph is greater than 64, *Phase2* creates two files: *AdjLst* and *ProcLst*. *AdjLst* is a list of the adjacencies of the graph with the port and program code information in an abbreviated form. *ProcLst* is a list of each of the node numbers and their corresponding procedure name. The first three lines of *ProcLst* are dedicated to: a "magic number" which internally represents the graph family type, the node count, and a number which indicates which member of the graph family the input graph is. For our tree example, *AdjLst* and *ProcLst* are given in Figure 4. In Prep-P, control is then passed to the contraction routine.

If the number of nodes is less than or equal to 64, *Phase2* generates 3 files: *SmallAdjLst*, *AdjLst* and *ProcLst*. *SmallAdjLst* is an adjacency list which will be used directly as input to the layout routine. For our shuffle-exchange example (Figure 3), *SmallAdjLst*, *AdjLst* and *ProcLst* are given in Figure 5. In Prep-P, if the graph has at most 64 nodes, the layout routine is called after Phase2. Upon completion of the layout routine, the *CHiPParams* and *SwitchSet* files have been created and the shellscript calls *genPN* and *genCN* to create the *PortNames* and *CodeNames* files. The *.x* files are compiled by the XX compiler into *.s* files, and the user may call poker in the directory where the *.s* program files and the *PortNames*, *CodeNames*, *CHiPParams* and *SwitchSet* files reside. In particular, if the graph has at most 64 nodes, no multiplexing is needed.

Phase1 and *Phase2* may be called separately by typing

/lib/cpp <filename> | Phase1
/lib/cpp <filename> | Phase2

This creates a file called *graph.c*. *AdjLst* and *ProcLst* (and *SmallAdjLst* if the graph has at most 64 nodes) are produced as a result of the compilation and execution of *graph.c*. To compile and execute *graph.c*, type

cc -g graph.c -lm -ll; a.out

```
                9
                127
                7
:1: ROOT        :1:(LSON,PARENT)3:(RSON,PARENT)2
:2: ANCES       :2:(LSON,PARENT)5:(RSON,PARENT)4:(PARENT,RSON)1
:3: ANCES       :3:(LSON,PARENT)7:(RSON,PARENT)6:(PARENT,LSON)1
  .               .
  .               .
  .               .
:63: ANCES      :63:(LSON,PARENT)127:(RSON,PARENT)126:(PARENT,LSON)31
:64: LEAF       :64:(PARENT,RSON)32
:65: LEAF       :65:(PARENT,LSON)32
  .               .
  .               .
  .               .
:126: LEAF      :126:(PARENT,RSON)63
:127: LEAF      :127:(PARENT,LSON)63
```

(a) (b)

Figure 4
ProcLst (a) and *AdjLst* (b) generated by *Phase2* for the tree.

```
:0: 1       7                                   :0: INEND
:1: 2 0     4                                   :1: INNER
:2: 1 3     2                                   :2: INNER
:3: 2       :0:(EXCH,EXCH)1                     :3: OUTEND
            :1:(SHUF,SHUF)2:(EXCH,EXCH)0
            :2:(SHUF,SHUF)1:(EXCH,EXCH)3:
            :3:(EXCH,EXCH)2
```

(a) (b) (c)

Figure 5
SmallAdjLst (a), *AdjLst* (b) and *ProcLst* (c) generated by *Phase2* for the shuffle-exchange.

The Contraction Phase

During the contraction phase, the graph (with > 64 nodes) represented by the *AdjLst* file is contracted to fit into the CHiP architecture. We call the nodes representing processes in the original graph **virtual PEs (vips)** and the nodes representing processors in the target CHiP architecture **real PEs (rps).** The contraction phase then produces a mapping from vips to rps. In poker, there are at most 64 rps. Note that for the contractions implemented, the contracted graph is a smaller member of the same graph family as the original graph (with the exception of the shuffle-exchange graph which is contracted into a 4-pin shuffle).

The program component of Prep-P which performs the contraction is called *contract*. *contract* produces output in the form of two files: *ConAdjLst* and *VipToRp*. *ConAdjLst* is the adjacency list of the contracted graph without the program code identifier and port name information; it consists of a list of adjacency lines, one for each vertex in the graph. If v is a vertex in the graph and $w_1,...,w_n$ are the vertices adjacent to v then the adjacency line for v in *ConAdjLst* has the following format:

:v: w_1 w_2 w_3 ... w_n

In particular, the first vertex is delimited by colons and the vertices adjacent to it (including the first) are separated by spaces. If no vertices are adjacent to v, then v will appear in *ConAdjLst* anyway as

:v:

Lines of this form appear if the graph has isolated nodes or if the contracted graph does not have exactly 64 nodes. *ConAdjLst* is composed of a sequence of adjacency lines in any order.

VipToRp is textually similar to *ConAdjLst* in format but gives the mapping of vips to rps generated by *contract*. For each rp, *VipToRp* has a line containing all the vips mapped onto the given rp in the following format:

:rp: vip_1 vip_2 vip_3 ...

Each vip (node in the original graph) must appear to the right of the last colon in exactly one line of *VipToRp* since the embedding generated by *contract* is well-defined.

contract can be called separately in the same directory where *AdjLst* resides by typing *contract*.

The Layout Phase

The embedding of the rps on a 64 PE CHiP lattice with corridor width 1 is done by the program component of Prep-P called *layout*. *layout* takes as input the file *ConAdjLst* (or the file *SmallAdjLst* if there are at most 64 nodes) and produces the following output:

1) A (default tty) message to the user giving information on the number of vertices and edges, and the maximum and total wirelength in the CHiP routing (produced by *layout)* and in the manhattan metric. In addition, the placement of the rps is printed. Each rp is denoted by an integer rp number. The rp number will be the same as the rp numbers (the numbers between colons) in *VipToRp* if the graph has more than 64 nodes, and will be the same as the node numbers if the graph has at most 64 nodes.

2) A file called *Placement* with the rp placement information.

3) Initialized *CHiPParams* and *SwitchSet* files to be used by poker.

The layout program may be used individually on an input file containing an adjacency list in the same format as *ConAdjLst*. This can be done by typing

layout <filename>

in the directory in which the adjacency list *<filename>* resides. If a layout for a CHiP architecture with corridor width >1 (<5) is desired, the user should type

layout -w n <filename>

where *n* is the desired corridor width. (Although the corridor width may be any integer constant in the layout program, poker permits a maximum corridor width of 4). If a graph cannot be laid out by the layout routine with the given corridor width, an error message will appear in the output; however, the placement of the graph will still be given.

The layout program only accepts undirected graphs so if w_i appears in the adjacency line of v in the adjacency list of the graph, then v must appear in the adjacency line of w_i. Currently for our program, the number of nodes in the graph must be a power of 4. The layout program can place any graph but only graphs which can be laid out with a constant corridor width can be routed. Duplication of nodes in a line of the adjacency list (e.g. :1: 2 2) will cause an infinite loop.

After executing the layout program, the user can see the graph laid out on a 64 PE grid by calling poker from the directory which contains the *CHiPParams* and *SwitchSet* files (the current directory).

Premux

premux is called by Prep-P in the directory where *AdjLst, ProcLst, Placement, VipToRp* and the compiled *.s* files reside, and these files must be present for *premux* to execute. The premux routine concatenates the assembled XX program codes for all the vips to be mapped onto each PE, replaces the I/O calls (reads and writes) with our I/O routines, and initializes data areas to be used by the support routines. (Note that in Prep-P, the *.x* files have already been compiled by invoking the shellscript *XX *.x* which calls the *xx* compiler on all of the program code files and produces *.s* files).

premux produces 65 files: *TraceVar*, which contains the types of all trace variables in each PE to be used by *HxToO*, and *rpi,j.s* where $1 \leq i,j \leq 8$. *rpi,j.s* contains the concatenated source code for the real PE at location i,j. Code files for null PEs (PEs with no vips mapped onto them) are not generated. To call *premux* separately, just type

premux

SToA51

SToA51 is a shellscript which calls *supstoa51* on each of the *rpi,j.s* files and produces *rpi,j.a51* files suitable for assembling by *as51* (the standard Intel 8051 assembler). The standard program which converts source code to *as51* code is called *stoa51*. For clarity, we call this program *standstoa51*. *supstoa51* is a

modification of *stoa51 (standstoa51)* which includes information for linking support routines into the concatenated files. In particular, entry points and data areas are allocated and code space in the external RAM is reserved.

To call *STOA51* separately in the directory in which the *rpi.j.a51* files are located, type

STOA51 rp?.?.s

As51

As51 is a shellscript which calls *as51* (the standard intel 8051 assembler) on all *rpi.j.a51* files. The output is hex *rpi.j.hx* files. To call *As51*, type

As51 rp?.?.a51

in the directory in which the *rpi.j.a51* files reside.

Postmux

postmux is called in the directory in which the *rpi.j.hx* files and *TraceVar* reside. The output of *postmux* is *pei.j.hx* files. These are the rp hex files linked with the support routines needed for multiplexing the concatenated code for each PE. Trace variable information is also included in the *pei.j.hx* files. In addition, hex files for null PEs are produced in *postmux*. To call *postmux* separately, just type

postmux

HxToO

HxToO is a shellscript which converts the hex files *pei.j.hx* into object files *pei.j.o*. At the completion of *HxToO* in Prep-P, all the necessary assembled files are in the current directory along with the initialized *CHiPParams, PortNames, CodeNames* and *SwitchSet* files. To call *HxToO* separately, type

HxToO pe?.?.hx

in the directory where the *pei.j.hx* files reside.

Upon the completion of *HxToO*, Prep-P also terminates. At this time, lots of extra files are in the current directory, and before Prep-P relinquishes control to the user, all *rpi.j.a51, rpi.j.hx* and *rpi.j.s* files are removed. The user can now call poker.

The Prep-P I/O System

The most fundamental difference between the code produced by Prep-P and standard assembled code for poker is that the read and write routines for Prep-P must accomodate reads and writes between vips mapped onto the same rp and vips mapped onto different rps. Consequently, the read and write

routines for Prep-P are substantially more complex than the read and write routines in poker. Prep-P keeps all the different reads and writes straight by keeping a multipurpose array called *buffer* with message space for each port of each vip, and an array called *vip status* which gives the current state of each vip. Also of interest is the *buffer status* byte in the message area of each vip port in *buffer*.

In this section, we briefly describe the protocols used for inter- and intra-reads and writes by Prep-P and outline the function of the I/O handler. In a read, we will denote the vip who wishes to read as the *consuming* vip and the vip who must send the message to be read as the *producing* vip. Similarly, in a write, we will denote the vip who wishes to write as the *producing* vip and the vip who will be written to as the *consuming* vip. The following paragraphs outline Prep-P's behavior on intra- and inter- reads and writes in the concatenated user routines.

Intra-Reads

When one vip wishes to read from another vip mapped onto the *same* rp, an intra-read occurs. During the intra-read, all activity focuses on the consuming vip. The consuming vip first checks its buffer status. If the buffer status is *full*, the message is contained in the consuming vip's buffer and the consuming vip takes the message from the buffer, stores it in the reading variable (variable x in a XX *x*<-*A* read instruction), and sets the producing vip's vip status to *ready*. If the buffer status is not *full*, the consuming vip stores the address of the reading variable in the buffer, sets the buffer status to *hungry*, and sets the vip status of the consuming vip to *waiting*. After either branch, there is a jump to the I/O handler.

Inter-Reads

When one vip wishes to read from another vip mapped into a *different* rp, an inter-read occurs. To perform an inter-read, the consuming vip must send a *data request* to the producing vip. The consuming vip then sets its vip status to *waiting*, puts the reading variable address in the buffer, and jumps to the I/O handler.

Intra-Writes

When one vip wants to write to another vip mapped to the *same* rp, an intra-write occurs. To perform an intra-write, the buffer status of the consuming vip is checked first. If the status is *hungry*, the producing vip puts the message into the address stored in the consuming vip's buffer, the buffer status of the consuming vip is set to *empty*, the vip status of the consuming vip is set to *ready*, and there is a jump to the I/O handler. If the consuming buffer status is not *hungry*, the message is stored in the consuming vip's buffer, the consuming vip's buffer status is set to *full*, the producing vip's vip status is set to *wait*, and there is a jump to the I/O handler.

Inter-Writes

When one vip wants to write to another mapped onto a *different* rp, an inter-write occurs. To perform an inter-write, the buffer status of the producing vip is checked first. If the status is *hungry*, the message is sent to the consuming vip's rp, the producing vip's buffer status is set to *empty*, and there is a jump to the I/O handler. If the buffer status of the producing vip is not *hungry*, the message is stored in the producing vip's buffer. The vip status of

the producing vip is then set to *waiting*, and there is a jump to the I/O handler.

The I/O Handler

The The I/O handler proceeds as follows: It first checks to see if a message has come in on any rp port. If so, the entire message is read. The message will either be a *data request* or a *data transmission*.

If the message is a *data request* (sent by an inter-read), the buffer of the producing vip is checked. If the buffer status is *full*, then the message stored in the producing vip's buffer is sent out in the direction the request came from, the producing vip's buffer status is set to *empty*, and the producing vip's vip status is set to *ready*. If the producing vip's buffer status is not *full*, the producing vip's buffer status is set to *hungry* and the direction of the *data request* is stored in the producing vip's buffer.

If the message recieved by the I/O handler is a *data transmission* (in response to an inter-read), then the data field of the message is stored in the address given in the consuming vip's buffer, the consuming vip's buffer status is set to *empty*, and the consuming vip's vip status is set to *ready*.

After either a *data transmission* or a *data request*, the I/O handler loops until there are no more messages. When there are no more messages, the I/O handler jumps to the context switcher.

The Context-Switcher

The context switcher is jumped to from the I/O handler. It first saves the environment (stack, important registers, etc.) of the current vip. Next, it does a wrap-around search of the vip status array (starting from the current vip) for the next vip whose vip status is *ready*. If a vip whose vip status is *ready* is found, the environment for that vip is loaded into memory and control is transfered to the point in that vip's code where it was interrupted. If all of the vips have their vip status set to *dead*, the program halts. (The vip status of a vip is set to *dead* when the vip code has completed execution). If no vip has vip status *ready*, and some vip is still waiting on I/O (has vip status not set to *dead*), the context switcher jumps to the I/O handler.

Trace Variables in Prep-P

Poker only allows the user to follow 4 trace variables through the execution of the code [S2]. In the *postmux* portion of Prep-P, we have added support routines so that the tracing programs for poker can be used with our concatenated codes. After running Prep-P, the first trace variable is devoted to the vip number. In particular, this tells the user which vip is executing when the program stops. This first variable is assigned by *postmux* and should not be included in the explicit trace code added to the source programs by the user. The next three variables are the first 3 (explicit) trace variables from the concatenated code.

As an example, say PE i,j contains the concatenated codes for vip_1, vip_2, vip_3 and vip_4 in the given order. Assume that the XX program code for vip_1 includes trace variable A, the XX program code for vip_2 includes no trace

variables, the XX program code for vip_3 includes trace variables B and C and the XX program code for vip_4 includes trace variable D. Then the trace variables printed out for PE i,j will be the first three trace variables in the concatenated list: A, B and C. The trace variable D will not be shown.

The order in which the code is concatenated at each PE is always the order given in VipToRp. The mapping from single rp numbers to the 2 digit PE numbers appears on the terminal after the layout portion of Prep-P. This mapping can also be found in the *Placement* file where the first column of 4 digit numbers represents the PE (i.e. 0102 represents PE 1,2) and the second column of 2 digit numbers represents the rp number. For example, a line in *Placement* in which the first two columns were

0102 43 ...

would mean that rp number 43 is located at PE 1,2.

Appendix: Standard Numberings and Input Formats for Prep-P Graph Families

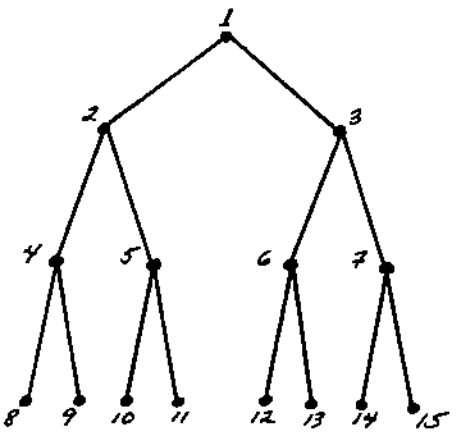
The following are input formats for the graph families which can be handled by the preprocessor. In the next version of the preprocessor, we are planning on using a more general contraction method to eliminate the restrictions on graph type and input format.

Complete Binary Tree

The standard numbering for complete binary trees is shown in Figure 6b). We let the number of the root be 1, and for each internal node i in the graph, we let the number of the left son be 2i and the number of the right son be 2i+1. A parallel algorithm using complete binary trees will typically use distinct program codes for the root, non-root internal nodes, and leaves. A sample graph specification format for such programs can be seen in Figure 6a). Note that the *nodecount* must always be 2^k-1 for k> 0.

```
#define nc 2^k-1
tree
nodemin = 1
nodecount = nc
procedure ROOT
   nodetype: {i == 1}
   port RSON: {2*i}
   port LSON: {2*i+1}
procedure ANCES
   nodetype: {i> 1 && i≤ nc/2}
   port PARENT: {i/2}
   port RSON: {2*i}
   port LSON: {2*i+1}
procedure LEAF
   nodetype: {i> nc/2}
   port PARENT: {i/2}
```

a)



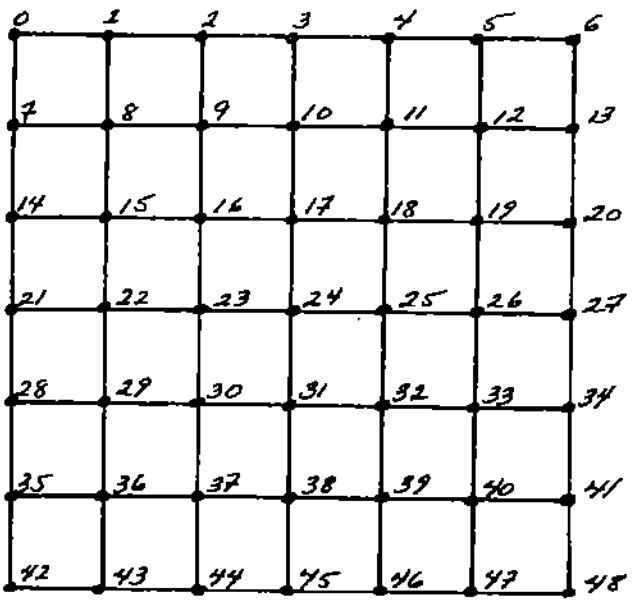
b)

Figure 6
Graph specification and numbering for a complete binary tree.

Square Mesh

We use the row major numbering for square meshes shown in Figure 7b). (Let the number of the left-hand top corner be 1. Number consecutively across the top row and continue at the beginning (left end) of the second row.) A parallel algorithm using square meshes may use distinct program codes for the corners, for the left, right, upper and lower border nodes, and for the internal nodes. A sample graph specification format for such programs is shown in Figure 7a). The *nodecount* for a square mesh will always be k^2 for $k>0$.

```
#define n k
mesh
nodemin = 0
nodecount = n*n
procedure NWCORNER
   nodetype: {i == 0}
   port EAST: {1}
   port SOUTH: {n+i}
procedure NECORNER
   nodetype: {i == n-1}
   port WEST: {n-2}
   port SOUTH: {2n-1}
procedure SECORNER
   nodetype: {i == n*n-1}
   port WEST: {n*n-2}
   port NORTH: {n*(n-1)-1}
procedure SWCORNER
   nodetype: {i == n*(n-1)}
   port EAST: {n*(n-1)+1}
   port NORTH: {n*(n-2)}
procedure NBORDER
   nodetype: {i> 0 && i< n-1}
   port WEST: {i-1}
   port EAST: {i+1}
   port SOUTH: {i+n}
procedure SBORDER
   nodetype: {i> n*(n-1) && i< n*n-1}
   port WEST: {i-1}
   port EAST: {i+1}
   port NORTH: {i-n}
procedure WBORDER
   nodetype: {i> 0 && i< n*(n-1) && i%n == 0}
   port NORTH: {i-n}
   port SOUTH: {i+n}
   port EAST: {i+1}
procedure EBORDER
   nodetype: {i> n-1 && i< n*n-1 && i%n == n-1}
   port NORTH: {i-n}
   port SOUTH: {i+n}
   port WEST: {i-1}
procedure INTERIOR
   nodetype {i> n-1 && i< n*(n-1) && i%n> 0 && i%n< n-1}
   port NORTH: {i-n}
```



```
port SOUTH: {i+n}
port EAST: {i+1}
port WEST: {i-1}
```

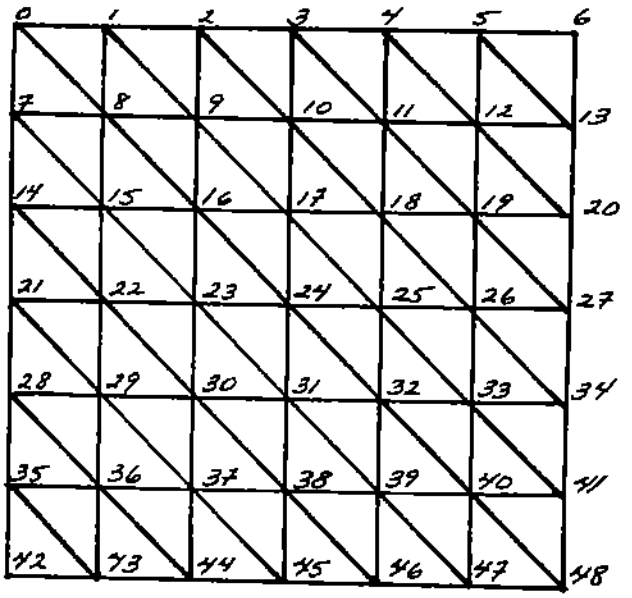
a) b)

Figure 7
Graph specification and numbering for a square mesh.

Hexagonal Mesh

The hexagonal mesh has the same standard numbering and *nodecount* as the square mesh. In particular, the *nodecount* for a hexagonal mesh will always be k^2 for $k>0$.

```
#define n k
hexmesh
nodemin = 0
nodecount = n*n
procedure NWCORNER
   nodetype: {i == 0}
   port EAST: {1}
   port SOUTH: {n+i}
   port SE: {n+i+1}
procedure NECORNER
   nodetype: {i == n-1}
   port WEST: {n-2}
   port SOUTH: {2n-1}
   port SW: {2n-2}
procedure SECORNER
   nodetype: {i == n*n-1}
   port WEST: {n*n-2}
   port NORTH: {n*(n-1)-1}
   port NW: {n*(n-1)-2}
procedure SWCORNER
   nodetype: {i == n*(n-1)}
   port EAST: {n*(n-1)+1}
   port NORTH: {n*(n-2)}
   port NE: {n*(n-2)+1}
procedure NBORDER
   nodetype: {i> 0 && i< n-1}
   port WEST: {i-1}
   port EAST: {i+1}
   port SOUTH: {i+n}
   port SW: {i+n-1}
   port SE: {i+n+1}
procedure SBORDER
   nodetype: {i> n*(n-1) && i< n*n-1}
   port WEST: {i-1}
   port EAST: {i+1}
   port NORTH: {i-n}
   port NW: {i-n-1}
   port NE: {i-n+1}
procedure EBORDER
   nodetype: {i> 0 && i< n*(n-1) && i%n == 0}
```



```
    port NORTH: {i-n}
    port SOUTH: {i+n}
    port WEST: {i-1}
    port NW: {i-n-1}
    port SW: {i+n-1}
 procedure WBORDER
    nodetype: {i> n-1 && i< n*n-1 && i%n == n-1}
    port NORTH: {i-n}
    port SOUTH: {i+n}
    port EAST: {i+1}
    port NE: {i-n+1}
    port SE: {i+n+1}
 procedure INTERIOR
    nodetype {i> n-1 && i< n*(n-1) && i%n> 0 && i%n< n-1}
    port NORTH: {i-n}
    port SOUTH: {i+n}
    port EAST: {i+1}
    port WEST: {i-1}
    port NW: {i-n-1}
    port NE: {i-n+1}
    port SW: {i+n-1}
    port SE: {i+n+1}
```

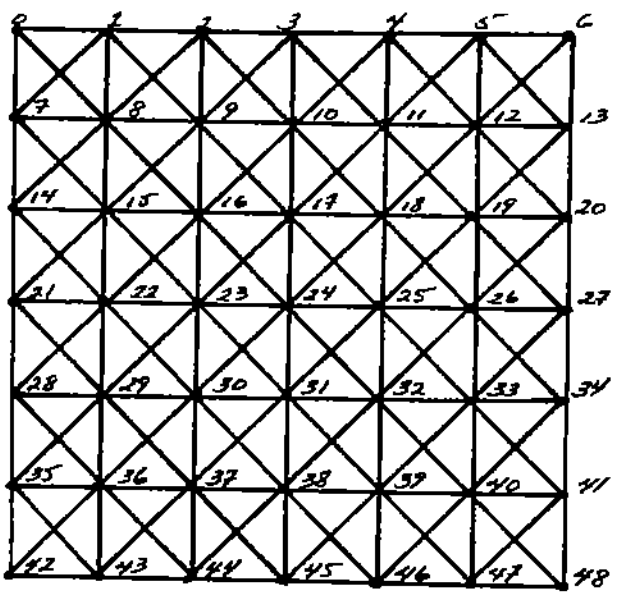
a) b)

Figure 8
Graph specification and numbering for the hexagonal mesh.

Finite Element Graph

The finite element graph has the same standard numbering and *nodecount* as the square mesh. In particular, the *nodecount* for a finite element graph will always be k^2 for $k > 0$.

```
#define n k
felem
nodemin = 0
nodecount = n*n
procedure NWCORNER
   nodetype: {i == 0}
   port EAST: {1}
   port SOUTH: {n+i}
   port SE: {n+i+1}
procedure NECORNER
   nodetype: {i == n-1}
   port WEST: {n-2}
   port SOUTH: {2n-1}
procedure SECORNER
   nodetype: {i == n*n-1}
   port WEST: {n*n-2}
   port NORTH: {n*(n-1)-1}
   port NW: {n*(n-1)-2}
procedure SWCORNER
   nodetype: {i == n*(n-1)}
   port EAST: {n*(n-1)+1}
```



```
   port NORTH: {n*(n-2)}
procedure NBORDER
   nodetype: {i> 0 && i< n-1}
   port WEST: {i-1}
   port EAST: {i+1}
   port SOUTH: {i+n}
   port SE: {i+n+1}
procedure SBORDER
   nodetype: {i> n*(n-1) && i< n*n-1}
   port WEST: {i-1}
   port EAST: {i+1}
   port NORTH: {i-n}
   port NW: {i-n-1}
procedure EBORDER
   nodetype: {i> 0 && i< n*(n-1) && i%n == 0}
   port NORTH: {i-n}
   port SOUTH: {i+n}
   port WEST: {i-1}
   port NW: {i-n-1}
procedure WBORDER
   nodetype: {i> n-1 && i< n*n-1 && i%n == n-1}
   port NORTH: {i-n}
   port SOUTH: {i+n}
   port EAST: {i+1}
   port SE: {i+n+1}
procedure INTERIOR
   nodetype {i> n-1 && i< n*(n-1) && i%n> 0 && i%n< n-1}
   port NORTH: {i-n}
   port SOUTH: {i+n}
   port EAST: {i+1}
   port WEST: {i-1}
   port NW: {i-n-1}
   port SE: {i+n+1}
```

a) b)

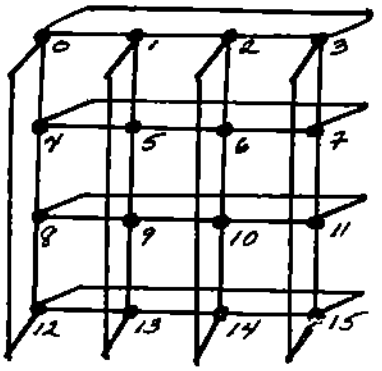
Figure 9
Graph specification and numbering for the finite element graph.

Torus

The torus has the same standard numbering and *nodecount* as the square mesh. Since the torus is a wrap-around mesh, parallel algorithms on the torus may have the same program code for each node. A sample graph specification for such algorithms is given in Figure 10a) Note that the *nodecount* for a torus will always be k^2 for $k > 0$.

```
#define n k
torus
nodemin = 0
nodecount = n*n
procedure NODE
  nodetype {i≥0 && i<n*n}
  port NORTH: {i>n-1 ? i-n : n*(n-1)+i}
  port SOUTH: {i<n*(n-1)-1 ? i+n : i%n}
  port EAST: {i%n<n-1 ? i+1 : i-n}
  port WEST: {i%n>0 ? i-1 : i+n}
```

a)



b)

Figure 10
Graph specification and numbering for the torus.

Line

The line has a standard numbering which starts at 1 on one end and numbers the nodes consecutively along the line as in Figure 11b). Parallel algorithms for linear systolic arrays may have distinct programs for each end of the array and for the internal nodes in the array. A sample specification for such algorithms is shown in Figure 11a).

```
#define nc k
line
nodemin = 0
nodecount = nc
procedure FIRST
  nodetype: {i == 0}
  port NEXT: {i == 1}
procedure MIDDLE
  nodetype: {i>0 && i<nc-1}
  port NEXT: {i+1}
  port PREV: {i-1}
procedure LAST
  nodetype: {i == nc-1}
  port PREV: {i-1}
```

a)

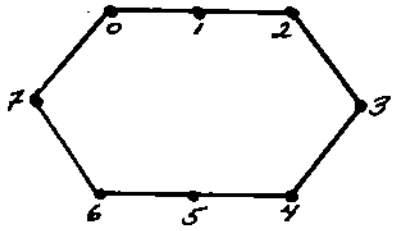
b)

Figure 11
Graph specification and numbering for the line.

Loop

The loop (or ring) has the same numbering and *nodecount* as the line. Many parallel algorithms on a ring use the same program code at each of the nodes. A sample graph specification for such algorithms is shown in Figure 12a).

```
#define nc k
loop
nodemin = 0
nodecount = nc
procedure NODE
  nodetype: {i≥ 0 && i≤ nc-1}
  port NEXT: {i<nc-1 ? i+1 : 0}
  port PREV: {i>0 ? i-1 : nc-1}
```



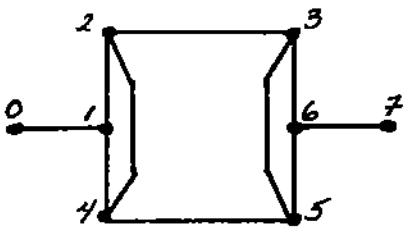
a) b)

Figure 12
Graph specification and numbering for the loop.

Shuffle-Exchange

A shuffle-exchange graph consists of 2^j nodes. In the standard numbering, each node is labelled by a binary j-bit string. Adjacent nodes have labels which are either left or right circular shifts of one another (the shuffle edges), or have labels in which the last bit is complemented (the exchange edges). A shuffle-exchange graph with 8 nodes is shown in Figure 13b). Parallel algorithms on a shuffle-exchange graph often use a distinct program code at each of the degree 1 nodes (0^j and 1^j), and a distinct program code at the inner nodes. A sample specification for such programs is given in Figure 13a) if j>2. If j=2, a sample specification is given in Figure 3a).

```
#define m 2^(j-1)+2^(j-3)+...+2^0 or 1
#define k j
#define n 2^j
shuffexch
nodemin = 0
nodecount = n
procedure INEND
  nodetype: {i == 0}
  port EXCH: {1}
procedure INNER
  nodetype: {k%2!=0 && i>0 && i<n-1}
  port EXCH: {i%2==0 ? i+1 : i-1}
  port RSHUF: {i%2==0 ? i/2 : (i-1)/2 + n/2}
  port LSHUF: {i>n ? (i-n)*2+1 : 2*i}
procedure INNER
  nodetype: {k%2==0 && i>0 && i<n-1 && i!=m && i!=(m-1)/2}
  port EXCH: {i%2==0 ? i+1 : i-1}
  port RSHUF: {i%2==0 ? i/2 : (i-1)/2 + n/2}
  port LSHUF: {i>n ? (i-n)*2+1 : 2*i}
procedure MIDDLE
  nodetype: {k%2==0 && i==m}
  port EXCH: {i%2==0 ? i/2 : (i-1)/2 + n/2}
  port SHUF: {m/2}
procedure MIDDLE
  nodetype: {k%2==0 && i==m/2}
  port EXCH: {i%2==0 ? i/2 : (i-1)/2 + n/2}
  port SHUF: {m}
```



```
procedure OUTEND
  nodetype: {n-1}
  port EXCH: {n-2}
```

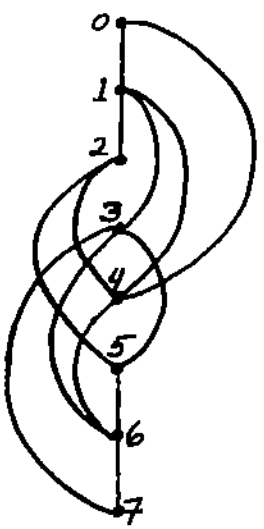
a) b)

Figure 13
Graph specification and numbering for the Shuffle-Exchange.

4-pin Shuffle

The 4-pin shuffle is the image under our contraction of a shuffle-exchange graph. It is a shuffle-exchange graph in which the incident nodes for each exchange edge have been coalesced into one. The 4-pin shuffle on 8 nodes is illustrated in Figure 14b). Since the 4-pin shuffle has no exchange edges, a parallel algorithm on the 4-pin shuffle can be represented by the simplified shuffle-exchange code given in 14a). The *nodecount* for the 4-pin shuffle must be 2^j where $j > 0$.

```
#define m 2^(j-1)+2^(j-3)+...+2^0 or 1/2
#define k j
#define n 2^(j-1)
s4pin
nodemin = 0
nodecount = 2n
procedure MIDDLE
  nodetype: {k%2==0 && i==m}
  port IN0: {i/2}
  port IN1: {2*i-2n+1}
  port OUT1: {i/2+n}
procedure MIDDLE
  nodetype: {k%2==0 && i==(m-1)/2}
  port IN0: {i/2}
  port IN1: {2*i}
  port OUT1: {2*i+1}
procedure ENDS
  nodetype: {i==0 || i==1}
  port OUT: {i==0 ? 1 : 2n-2}
  port IN: {i==0 ? n : n-1}
procedure NODE
  nodetype: {k%2!=0 && i≥0 && i<2n}
  port OUT0: {i<n ? 2*i : 2*i-2n}
  port OUT1: {i<n ? 2*i+1 : 2*i-2n+1}
  port IN0: {i/2}
  port IN1: {i/2+n}
procedure NODE
  nodetype: {k%2==0 && i≥0 && i<2n && i!=m && i!=(m-1)/2}
  port OUT0: {i<n ? 2*i : 2*i-2n}
  port OUT1: {i<n ? 2*i+1 : 2*i-2n+1}
  port IN0: {i/2}
  port IN1: {i/2+n}
```



a) b)

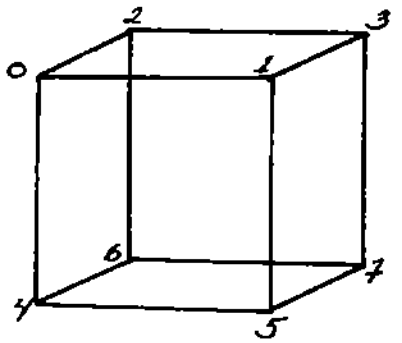
Figure 14
Graph specification and numbering for the 4-pin shuffle.

Binary n-cube

The binary n-cube consists of 2^k nodes. Each node is labelled by a k-bit binary number like the shuffle-exchange graph. However in the binary n-cube, there is an edge between any two vertices who differ in exactly one place by one bit (e.g. there is an edge between 00101 and 00001). Algorithms on the binary n-cube are likely to use the same program code for each of the nodes. A sample specification for such algorithms is given in Figure 15a). Since each node has degree log k, note that there is no uniform graph specification which can be instantiated by *#defines*.

```
bincube
nodemin = 0
nodecount = 2^k
procedure NODE
  nodetype: {i≥0 && i<2^k}
  port PORT1: {i&1==0 ? i+1 : i-1}
  port PORT2: {i&2==0 ? i+2 : i-2}
  port PORT3: {i&4==0 ? i+4 : i-4}
      .
      .
      .
  port PORTk-1: {i&2^k-1 ? i+2^k-1 : i-2^k-1}
```

a)



b)

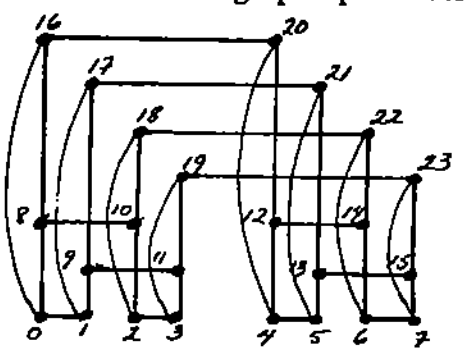
Figure 15
Graph specification and numbering for the binary n-cube.

Cube-Connected Cycle

The cube-connected cycle (CCC) is a network used to model the binary n-cube by a graph with bounded degree. The *nodecount* of a CCC is $k2^k$ where $k>0$. To label the nodes, we give each node a number computed from the cycle number (numbered across the "bottom" of the figure from the bottom leftmost corner) and the position number on the cycle (numbered from 0 upwards) times 2^k. Such a numbering is pictured in Figure 16b). Since the CCC models the binary n-cube, many parallel algorithms on the CCC have only one distinct program code for the nodes. Such a graph specification is given in Figure 16a).

```
#define m 2^k
#define j k
#define nc k*m
ccc
nodemin = 0
nodecount = nc
procedure NODE
  nodetype: {i≥0 && i<nc}
  port FWD: {i<(j-1)m ? i+m : i%m}
  port BACK: {i<m ? (j-1)m+i : i-m}
  port ACROSS: {i&(1<<(i/m))==0 ? i+(1<<(i/m)) : i-(1<<(i/m))}
```

a)



b)

Figure 16
Graph specification and numbering for the CCC.

Acknowledgements

We are grateful to many people for their help on this project. In particular, we have recieved encouragement and support from Dennis Gannon, Kevin Smallwood, Larry Snyder, John Rice and all the people on the Blue CHiP project. In particular, this project would have been immensely more difficult if not for the help and support of Steve Holmes and Ko-Yang Wang.

Bibliography

[KR] Kernighan, B. and D. Ritchie, *The C Programming Language*, Prentice Hall, 1978.

[S1] Snyder, L., "Introduction to the Configurable, Highly Parallel Computer," Computer, 15(1): 47-56, January, 1982.

[S2] Snyder, L., "The Poker (1.1) Programmers Guide," Technical Report CSD-TR-434, Purdue University 1983.

[S3] Snyder, L., Albert, S., Amport, C., Beuning, B., Chester, A., Guaragno, J., Kent, C., Love, J., Shekita, E. and C. Smith, "The Poker Programming Environment and Its Implementation," Technical Report CSD-TR-410, Purdue University, 1982.